



MAITRISEZ LE CONTROLE DE VERSION ET LA COLLABORATION (GIT & GITHUB)

Guide

Résumé

Cet article permettra son lecteur d'avoir les prérequis sur le contrôle de version avec git et GitHub :

- Connaître comment créer un dépôt git
- Connaître certaines notions avancées de git
- Connaître comment collaborer sur GitHub...

Daryl Noupik
daryloupik@gmail.com

Nous allons commencer notre cours par une petite histoire :

Un jour, dans le royaume numérique de CodeLand, il y avait deux développeurs, Diana et Teemo, qui travaillaient sur un projet magique appelé "WizardApp". Diana avait une idée brillante pour ajouter une nouvelle fonctionnalité, tandis que Teemoo avait repéré un bug mystérieux.

Sans utiliser Git et GitHub, ils décidèrent de travailler directement sur le même fichier, comme dans les temps anciens. Diana ajouta sa fonctionnalité, tandis que Teemoo traquait le bug. À la fin de la journée, ils réalisèrent qu'ils avaient tous deux apporté des modifications au même endroit du fichier.

Teemo s'exclama, "Oh non, j'ai écrasé ta fonctionnalité avec ma correction de bug !"

Diana soupira, "Et maintenant, le bug est toujours là, et la fonctionnalité a disparu dans les abysses du code perdu."

C'est alors qu'un vieux développeur sage, Aphelios, apparut. Il leur dit : "Hé, les amis, vous auriez dû utiliser Git et GitHub. C'est comme une baguette magique pour la collaboration !"

Intrigués, Diana et Teemo suivirent les conseils de Aphelios et créèrent un dépôt WizardApp sur GitHub. Ils commencèrent à travailler sur des branches séparées pour leurs idées. Lorsqu'Diana eut terminé sa fonctionnalité et Teemoo son correctif, **ils créèrent des pull requests**.

Aphelios expliqua : "Maintenant, vous pouvez fusionner vos idées sans écraser le travail de l'autre. Et si quelque chose tourne mal, Git vous permet de revenir en arrière dans le temps, comme des sorciers du code !"

Diana et Teemoo suivirent les conseils de Aphelios, et WizardApp devint le projet le plus magique de CodeLand. Désormais, chaque développeur du royaume savait que Git et GitHub étaient les vraies clés pour éviter les catastrophes de code et rendre la collaboration aussi lisse que des sorts bien lancés. Et ils vécurent tous heureux et collaborèrent joyeusement pour le reste de leurs jours.

Table of Contents

Introduction :	5
I. Comprendre GIT	5
A. Les fondamentaux	5
1) Installation	5
2) Configuration de Git	6
II. Concepts Cles de git	7
Enregistrer des modifications sur un dépôt	9
Vérifier l'état des fichiers	9
Placer de nouveaux fichiers sous de version	10
Ignorer des fichiers	10
Inspecter les modifications indexées et non indexés	11
Effacer des fichiers	11
Déplacer des fichiers	12
Visualiser l'historique des validations	12
Annuler des actions	12
Désindexer un fichier déjà indexé	13
Afficher des dépôts distants	13
Ajouter des dépôts distants	14
Pousser son travail sur un dépôt distant	14
Inspecter un dépôt distant	14
Retirer et renommer des dépôts distants	15
Fonctionnalistes avancées	15
Etiquetage	15
Créer des étiquettes	15
Partager les étiquettes	16
Supprimer les étiquettes	16
Les branches	17
Créer une nouvelle branche	17
Basculer vers une branche existante	17
Créer une nouvelle branche et basculer directement	18
Fusionner des branches	18
Renommer une branche	18

Supprimer une branche locale	18
Supprimer une branche distante	18
Créer une nouvelle branche à partir d'un commit spécifique	19
Récupérer les modifications du dépôt distant.....	19
Récupérer les modifications sans fusionner	19
Retourner à un état antérieur du projet.....	19
Annuler les modifications locales.....	20
Annuler les modifications apportées par un commit.....	20
Mettre de côté les modifications locales	20
Retourner à un état antérieur du projet.....	21
Afficher les différences entre deux branches :	21
Réorganiser l'historique des commits	21
Fusionner un commit spécifique	22
Restaurer un fichier depuis un commit spécifique avec un dialogue interactif	22
Afficher les différences dans un commit.....	22
Ignorer les espaces lors de la fusion (merge)	22
Générer un rapport de bogue.....	22
Voir les révisions et l'auteur des changements	23
Git rebase Vs Git reset Vs git revert.....	23
Qu'est-ce que GitHub ?	23
Fonctionnalités principales de GitHub.....	24
Pull requests	24
Comment créer un pull request	24
Comment approuver un pull request.....	24
Comment rejeter un pull request.....	24
Forks	25
Comment créer un fork.....	25
Comment travailler sur un fork.....	25
Les issues	26
Comment créer une issue	26
Comment suivre une issue.....	26
Comment résoudre une issue	26
Types d'issues	27

Autres fonctionnalités de GitHub 27

Introduction :

Le contrôle de version est un élément essentiel du développement logiciel moderne, permettant aux équipes de travailler de manière collaborative, de suivre les changements de code et de revenir à des versions antérieures si nécessaire. Git et GitHub sont des outils clés dans cet écosystème, offrant une solution robuste pour le suivi des versions et la collaboration en équipe. Cet article vise à explorer en détail Git et GitHub, depuis les concepts de base jusqu'aux fonctionnalités avancées.

I. Comprendre GIT

A. Les fondamentaux

Git est un système de contrôle de version distribué gratuit et open source conçue pour gérer tout, des petits aux très grands projets, avec rapidité et efficacité (Il s'agit donc d'un outil de développement qui aide une équipe de développeurs à gérer les changements apportés au code source au fil du temps). Créée en 2005 par Linus Torvalds et actuellement à sa version **2.42.1 (13/11/2023)**.

Facile à apprendre avec plusieurs avantages tel que :

- La collaboration facile
- Suivi des modifications
- Gestion des branches
- La rapidité
- Le Support solide de la communauté ...

1) Installation

Sous Windows :

Téléchargez et installez [Git pour Windows](#). Une fois installé, Git est disponible à partir de l'invite de commandes ou PowerShell.

Sous MacOS :

- Si Homebrew n'est pas déjà installé, suivez les instructions sur <https://brew.sh> pour l'installer.
- Ouvrez un terminal et exécutez la commande : **brew install git**.

Où

- Téléchargement depuis le site Web : - Rendez-vous sur <https://git-scm.com/download/mac> et téléchargez le programme d'installation.
- Exécutez le programme d'installation téléchargé et suivez les instructions.
- Vérification de l'installation : Dans le terminal, tapez **git --version** pour vérifier que Git a été installé avec succès.

Sous Linux :

- Utilisation d'un gestionnaire de paquets (Ubuntu/Debian) : - Ouvrez un terminal et exécutez la commande : **sudo apt update**.
- Ensuite, installez Git avec la commande : **sudo apt install git**.
- Utilisation d'un gestionnaire de paquets (Fedora) : - Ouvrez un terminal et exécutez la commande : **sudo dnf install git**.
- Utilisation d'un gestionnaire de paquets (openSUSE) : - Ouvrez un terminal et exécutez la commande : **sudo zypper install git**.

Dans le terminal tapez la commande **git --version**.

2) Configuration de Git

Une fois que vous avez installé Git sur votre système, vous voudrez personnaliser votre environnement Git. Ces paramètres seront qu'une seule fois et persisteront au fil des mises à jour (Vous pourrez les changer les changer au besoin en tapant les mêmes commandes).

Git contient un outil appelé **git config** qui vous permet de voir et modifier les variables de configuration qui contrôlent tous les aspects de l'apparence et du comportement de Git.

La liste des paramètres et leurs chemins d'accès est disponible via la commande :

```
$ git config --list --show-origin
```

Une fois git installer, il est nécessaire de définir votre identité en renseignant votre nom :

```
# Identity Name
> git config --global user.name "Darylis" (Dans l'invite de commande)
```

Et votre adresse mail

```
# Identity Email
> git config --global user.email "darylis@example.com"
```

L'option **--global** pour que cette modification affecte tous les dépôts avec lesquels vous travaillerez sur le system.

À présent que votre identité est renseignée, vous pouvez configurer l'éditeur de texte qui sera utilisé quand Git vous demande de saisir un message. Par défaut, Git utilise l'éditeur configuré au niveau système

```
# Editor Tool  
  
> git config --global core.editor subl
```

Par défaut Git crée une branche nommée **master** quand vous créez un nouveau repository avec git init. Depuis Git version 2.28, vous pouvez définir un nom différent pour la branche initiale.

Pour définir **main** comme nom de branche par défaut, utilisez la commande :

```
#Default branch  
  
$ git config --global init.defaultBranch main
```

L'on peut aussi définir l'outil de vérification des merges avec la commande :

```
# Diff Tool  
  
git config --global merge.tool filemerge
```

II. Concepts Cles de git

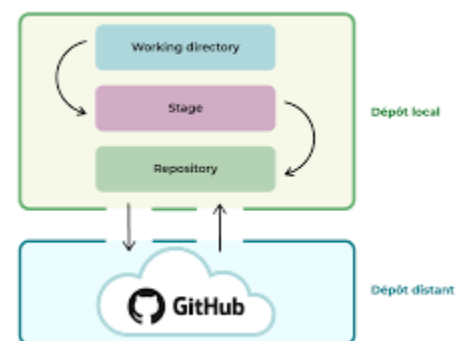
L'un des concepts clés de git est celui des dépôts ou repository (en anglais). L'on peut dire que :

Dépôt (Repository) : est un espace de stockage qui contient l'historique complet des modifications apportées à un projet, ainsi que les fichiers et dossiers actuels de ce projet. C'est une structure de données qui permet de suivre les différentes versions du code source d'un logiciel au fil du temps.

On distingue 2 types de dépôts :

Un dépôt local est une copie du projet Git stockée sur votre machine. Il contient l'historique complet des changements et vous permet de travailler hors ligne.

Un dépôt distant est une version du projet qui est hébergée sur un serveur, souvent sur une plate-forme comme GitHub. Les dépôts distants facilitent la collaboration en permettant à plusieurs développeurs de travailler sur le même projet et de partager leurs modifications.



Démarrage d'un projet Git

Une fois l'installation et la configuration de git effectuée afin de démarrer un projet git il est possible de le faire de deux manières :

- En transformant un répertoire existant en dépôt git
- En clonant un dépôt git existant sur un autre serveur (ou une autre machine)

Initialisation d'un dépôt Git dans un répertoire existant :

Pour ce faire il vous suffit de vous positionner dans le répertoire du projet. Si vous ne le l'avez jamais fait auparavant cela se présente ainsi selon votre système :

pour Linux:

```
$ cd /home/user/my_project
```

pour macOS:

```
$ cd /Users/user/my_project
```

pour Windows:

```
$ cd C:/Users/user/my_project
```

Vous validez et ensuite vous tapez la commande :

```
> git init
```

Cela crée un nouveau sous-répertoire nommé **.git** qui contient tous les fichiers nécessaires au dépôt — un squelette de dépôt Git.

```
C:\Users\COMPUTER CARE\OneDrive\Documents\Formation\HTML CSS SASS\HTML CSS>git init
Initialized empty Git repository in C:/Users/COMPUTER CARE/OneDrive/Documents/Formation/HTML CSS SASS/HTML CSS/.git/
```

Cloner un dépôt existant


Lorsque vous voulez une copie d'un dépôt Git existant. Par exemple, un projet auquel vous aimeriez contribuer ou tout simplement faire des tests en local. La commande dont vous avez besoin s'appelle **git clone**.

Git recevra une copie de quasiment toutes les données dont le serveur dispose. Toutes les versions de tous les fichiers pour l'historique du projet sont téléchargées quand vous lancez git clone

Vous clonez un dépôt avec **git clone [url]**. Par exemple les étudiants de première année de l'institut Ucac-Icam promo 2028 qui aimeraient avoir accès à la correction de leur devoir le jeu du pendu disponible sur mon profil pourront taper la commande suivante :

```
> git clone https://github.com/DarylNoupik/cpp-pendu-lab.git
```

Enregistrer des modifications sur un dépôt




« C'est super nous savons maintenant comment créer un dépôt ou utiliser un dépôt distant. Mais comment on l'utilise ? » vous vous demandez.

A présent que notre dépôt git est valide. Lorsque vous ferez une modification sur votre projet et validerez cette dernière, grâce à GIT il vous sera possible d'enregistrer cet état comme état suivi .

Chaque fichier de votre copie de travail peut avoir deux états : **sous suivi de version ou non suivi**. Les fichiers suivis sont les fichiers qui appartenaient déjà au dernier instantané ; ils peuvent être inchangés, modifiés ou indexés. En résumé, les fichiers suivis sont ceux que Git connaît.

Tous les autres fichiers sont non suivis



« Mais comment saurons-nous que nous n'avons pas enregistré les modifications dans notre dépôt pour commencer et que les fichiers sont suivis ou pas ? »

Vérifier l'état des fichiers

Git dispose d'un outil qui permet de déterminer l'état des fichiers qui n'est autre que la commande **git status**.

Git status : permet de vérifier l'état des fichiers.

En faisant un petit exemple sur mon pc voici un résultat possible :

```
C:\Users\COMPUTER CARE\OneDrive\Documents\Formation\HTML CSS SASS\HTML CSS>git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .vscode/
    a-propos.html
    css/
    images/
    index.html
    portfolio.html

nothing added to commit but untracked files present (use "git add" to track)
```

Placer de nouveaux fichiers sous de version

Une fois les fichiers non suivis détecter pour les placer sous le suivi de version ou les enregistrer il suffit de faire un :

```
>git add nom_fichier : pour un fichier en
particulier

Ou

>git add . : pour tous les fichiers non suivis.
```

On l'utilise aussi pour index les fichiers.

Ignorer des fichiers

Il arrive que certains fichiers ne doivent pas être suivis par Git. On peut les ignorer en énumérant leurs noms dans un fichier **.gitignore**

Cela permet d'éviter d'inclure des fichiers temporaires ou des dépendances dans le suivi de Git.

Ci-dessous un exemple de fichier .gitignore :

```
C:\Users\COMPUTER CARE\OneDrive\Documents\Formation\FrontendMentor\Solutions\product-preview-card-component-main>TYPE .gitignore
# pas de fichier .a
*.a

# mais suivre lib.a malgré la règle précédente
!lib.a

# ignorer uniquement le fichier TODO à la racine du projet
/TODO

# ignorer tous les fichiers dans le repertoire build
build/
```

Inspecter les modifications indexées et non indexés

La commande **git status** indique quels fichiers ont changé, mais pas ce qui a changé. La commande **git diff** montre les lignes exactes qui ont été ajoutées, modifiées ou effacées.

```
> git diff
```

Valider vos modifications

Pour valider des modifications, il faut les ajouter à l'index. Les fichiers qui ne sont pas dans l'index ne seront pas validés. Si vous avez déjà vérifié que tout est indexé, vous pouvez valider vos modifications

Une fois que vos fichiers ont bien été indexés vous pouvez passer à la validation des modifications avec la commande **git commit** selon la syntaxe suivante :

```
>git commit -m 'le_message_descriptif_de_la_modification'
```

Pour créer un commit, vous commencez par ajouter des fichiers modifiés à la zone de staging à l'aide de la commande **git add**. Ensuite, vous créez un commit en utilisant la commande **git commit -m "Message de commit"**. Le message de commit doit être descriptif, expliquant les changements apportés.

Le commit est ensuite soumis au dépôt local, enregistrant les modifications apportées aux fichiers.

Effacer des fichiers

Pour supprimer un fichier de Git, il faut le supprimer de l'index et valider. La commande **git rm** fait les deux. Si vous supprimez simplement le fichier, il réapparaîtra comme fichier non suivi à la prochaine validation.

```
> git rm le_nom_du_fichier
```

Si le fichier a été modifié, vous devez utiliser l'option **-f** pour forcer la suppression.

Si vous souhaitez supprimer le suivi de version d'un fichier sans le supprimer de votre copie de travail, utilisez l'option **--cached**.

```
> git rm --cached nom_fichier
```

Déplacer des fichiers

Git ne suit pas explicitement les mouvements de fichiers. Il détecte les renommages après coup.

La commande **git mv** permet de renommer un fichier dans Git. Git gère le renommage en détectant que le fichier a été déplacé.

```
> git mv nom_origine nom_cible
```

Visualiser l'historique des validations

Après avoir créé votre premier commit, vous pouvez utiliser la commande **git log** pour afficher l'historique des commits. Cela vous permet de visualiser les modifications apportées à votre projet au fil du temps.

Lorsque vous travaillez en équipe, la commande **git log** peut également vous aider à voir qui a fait quoi et quand. Cela peut être utile pour suivre les modifications apportées au projet et pour résoudre les conflits

```
> git log  
  
> git log --oneline --graph -all : Visualisez  
l'historique des commits sous forme de graphe simple,  
affichant toutes les branches
```

La commande **git log** affiche l'historique des commits par défaut, dans l'ordre chronologique inverse.

Pour afficher les différences entre chaque commit, utilisez l'option **-p**. Pour limiter la sortie à deux commits, utilisez l'option **-2**

```
> git log -p -2
```

Annuler des actions

Dans Git, il est possible d'annuler des modifications. Cependant, certaines annulations sont définitives et peuvent entraîner des pertes de données.

Une des annulations les plus courantes est de valider une modification sans ajouter tous les fichiers ou avec un message de validation incorrect. Dans ce cas, vous pouvez valider les modifications manquantes ou corriger le message de validation avec l'option **--amend**

```
> git commit --amend
```

Si aucune modification n'a été apportée depuis le dernier commit, l'instantané sera identique et la seule modification sera le message de validation.

Par exemple, si vous validez un commit puis réalisez que vous avez oublié d'indexer les modifications d'un fichier, vous pouvez utiliser la commande **git commit --amend** pour ajouter le fichier au commit.

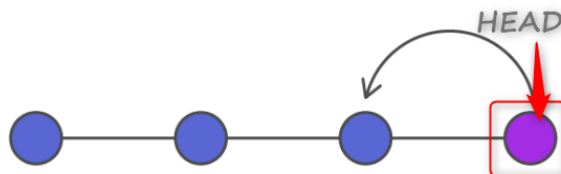
```
> git commit -m 'validation initiale'
> git add fichier_oublie
> git commit --amend
```

Désindexer un fichier déjà indexé

La commande **git status** permet de connaître l'état de la zone d'index et de la zone de travail.

Si vous avez accidentellement indexé un fichier que vous ne voulez pas valider, vous pouvez utiliser la commande **git reset HEAD <fichier>** pour le désindexer.

```
> git reset HEAD <fichier>
```



Afficher des dépôts distants

La commande **git remote** permet de lister les serveurs distants enregistrés.

Si vous avez cloné un dépôt, vous devriez voir au moins un serveur distant, nommé origin. Vous pouvez également utiliser l'option -v pour afficher l'URL du serveur distant.

```
> git remote -v
```

Ajouter des dépôts distants

Pour ajouter un nouveau dépôt distant Git, utilisez la commande **git remote add [nomcourt] [url]**.

Par exemple, pour ajouter un dépôt distant nommé pb à l'URL <https://github.com/paulboone/ticgit>, vous pouvez utiliser la commande suivante :

```
> git remote add dn https://github.com/DarylNoupik/tic-tac-toe  
Ou par défaut:  
git remote add origin https://github.com/DarylNoupik/tic-tac-toe
```

Pousser son travail sur un dépôt distant

Une fois les modifications validées. Pour pousser des modifications vers un dépôt distant, utilisez la commande **git push <nom-distant> <nom-de-branche>**:

```
> git push  
Ou pour envoyer vers une branche spécifique  
> git push origin MA_BRANCH.  
Ou encore si votre dépôt est lié à un dépôt distant et que vous souhaitez faire que Git configure la branche locale pour suivre la branche distante spécifiée en tant que branche distante par défaut.  
> git push -u origin ma_branche
```

Inspecter un dépôt distant

Pour obtenir des informations détaillées sur un dépôt distant, utilisez la commande **git remote show <nom-distant>**.

Par exemple, pour obtenir des informations détaillées sur le dépôt distant origin, vous pouvez utiliser la commande suivante :

```
> git remote show origin
```

Retirer et renommer des dépôts distants

Il peut arriver qu'au fil de l'eau vous souhaitez renommer ou retirer des dépôts distants.

Pour renommer un dépôt distant, utilisez la commande **git remote rename <nom-ancien> <nom-nouveau>**.

Par exemple, pour renommer le dépôt distant dn en Daryl, vous pouvez utiliser la commande suivante :

```
> git remote rename dn Daryl
```

Pour supprimer un dépôt distant, utilisez la commande **git remote rm <nom-distant>**.

Par exemple, pour supprimer le dépôt distant Daryl vous pouvez utiliser la commande suivante :

```
> git remote rm Daryl
```

Fonctionnalités avancées

Étiquetage

Git nous donne la possibilité d'étiqueter un certain état dans l'historique comme important. L'on utilise souvent cette fonctionnalité pour marquer les états de publication (le versionning : V1.0, V1.0.2 et ainsi de suite).

Créer des étiquettes

Git utilise deux principaux types d'étiquettes :

- **Les étiquettes légères** : sont simplement des pointeurs vers des commits. Elles ne contiennent aucune information supplémentaire, comme un message ou une signature.

Syntaxe de création :

```
> git tag v1.4-lg
```

- **Les étiquettes annotées** : sont des objets git complets. Elles ont une somme de contrôle, contiennent le nom et l'adresse e-mail du créateur, la date, un message d'étiquetage et peuvent être signées et vérifiées avec GNU Privacy Guard (GPG)

```
> git tag -a v1.4 -m 'ma version 1.4'
```

Afin de visualiser ou lister des tags vous pouvez utiliser respectivement les commandes : **git show** et **git tag**

```
> git show v1.4-lg  
> git tag
```

Partager les étiquettes

vous pouvez lancer **git push origin [nom-du-tag]**

```
> git push origin v1.5  
> git push origin --tags
```

Supprimer les étiquettes

Pour supprimer une étiquette de votre dépôt local, vous pouvez utiliser **git tag -d <nom-d-etiquette>**.

```
> git tag -d v1.4-lg
```

pour supprimer une étiquette distante utilise l'option **--delete** :

```
> git push origin --delete <nom-d-etiquette>
```

Les branches

Si maintenant vous souhaitez collaborer avec d'autres personnes et que vous avez divisé les tâches et les fonctionnalités de l'application à concevoir, il sera intéressant de créer des branches pour chacune d'entre elles pour les développer avant de les fusionner sur la branche principale

Les branches Git sont des points de référence dans l'historique d'un dépôt Git. Elles permettent de travailler sur des versions différentes du projet sans affecter l'historique principal

Créer une nouvelle branche

```
> git branch nom_de_la_branche
```

Basculer vers une branche existante

```
> git checkout nom_de_la_branche
```

Ou depuis la version 2.23 :

```
> git switch nom_de_la_branche
```

Cette commande permet de passer d'une branche à une autre et travailler sur des fonctionnalités isolées.

Créer une nouvelle branche et basculer directement

```
> git checkout -b nom_de_la_nouvelle_branche
```

Ou

```
> git switch -c nom_de_la_nouvelle_branche
```

Fusionner des branches

```
> git merge nom_de_la_branche
```

On l'utilise pour fusionner les changements d'une branche dans une autre, souvent pour intégrer une fonctionnalité terminée dans la branche principale.

Renommer une branche

```
> git branch -m nouveau_nom
```

Supprimer une branche locale

```
> git branch -d nom_de_la_branche
```

Contexte : Supprimez une branche locale après avoir fusionné les changements dans une autre branche.

Supprimer une branche distante

```
> git push origin --delete nom_de_la_branche
```

Contexte : Supprimez une branche distante. Assurez-vous d'avoir déjà supprimé la branche localement avant de supprimer la branche distante.

Créer une nouvelle branche à partir d'un commit spécifique

```
> git branch nouvelle_branche ID_du_commit  
> git checkout nouvelle_branche
```

Récupérer les modifications du dépôt distant

Contexte :

Vous travaillez sur un projet de développement logiciel avec d'autres personnes. Vous avez créé un dépôt Git pour le projet et vous travaillez sur la même branche que l'un de vos collègues. Votre collègue a effectué des modifications sur la branche et vous souhaitez les récupérer avant de continuer à travailler.

Solution

Pour récupérer les modifications de votre collègue, vous pouvez utiliser la commande git pull. Cette commande télécharge les modifications les plus récentes du dépôt distant et les fusionne dans votre branche locale.

Pour ce faire vous utiliserez la commande :

```
> git pull origin nom_de_la_branche
```

Récupérer les modifications sans fusionner

Maintenant si vous inspecter les changements avant de les fusionner :

```
> git fetch
```

Retourner à un état antérieur du projet

Vous souhaitez déplacer votre espace de travail vers un état précédent du projet en utilisant l'identifiant du commit.

```
> git checkout ID_du_commit
```

Annuler les modifications locales

Par la suite vous souhaitez annuler les modifications locales :

```
> git reset --hard
```

cela annule toutes les modifications locales non enregistrées. Attention, cela supprime définitivement les modifications non enregistrées.

- `--hard` : Supprime définitivement les modifications locales non enregistrées.
- `--soft` : Supprime les modifications locales non enregistrées, mais conserve les modifications enregistrées.
- `--mixed` : Supprime les modifications locales non enregistrées, mais conserve les modifications enregistrées et les fusionne avec la branche actuelle.

Annuler les modifications apportées par un commit

Git revert est une commande utilisée dans le système de contrôle de version Git pour annuler les modifications apportées par un commit spécifique. Elle crée essentiellement un nouveau commit qui inverse les modifications apportées dans le commit d'origine, annulant essentiellement ces modifications sans affecter l'historique global du projet.

```
> git revert <commit-hash>
```

La principale différence entre git revert et git reset est que git revert crée un nouveau commit qui inverse les modifications apportées par le commit spécifié, tandis que git reset supprime le commit spécifié de l'historique des commits.

Mettre de côté les modifications locales

Maintenant vous souhaitez mettre de côté temporairement vos modifications locales sans les committer. Pour car vous devez basculer entre les branches rapidement.

La commande :

```
> git stash
```

Ensuite pour les récupérer :

```
> git stash apply.
```

Retourner à un état antérieur du projet

Par la suite vous constaterez qu'il y a une erreur qui se glisse dans le code et cela a créé un problème :

```
> git bisect start  
> git bisect bad  
> git bisect good ID_du_commit
```

git bisect pour trouver quel commit a introduit un problème en effectuant une recherche binaire dans l'historique.

Afficher les différences entre deux branches :

```
> git diff branche_1..branche_2
```

Réorganiser l'historique des commits :

Contexte : Réorganisez l'historique des commits pour rendre l'historique plus linéaire. Utile pour garder un historique propre avant de fusionner des branches.

```
> git rebase branche_cible
```

Git rebase est une commande qui permet de réorganiser l'historique des commits. Cela signifie que vous pouvez déplacer des commits vers une autre branche, fusionner des branches de manière plus propre, ou simplement nettoyer l'historique de votre projet.

Fusionner un commit spécifique

```
> git cherry-pick ID_du_commit
```

Contexte : Appliquez sélectivement un commit spécifique sur votre branche actuelle. Utile lorsque vous souhaitez intégrer sélectivement des modifications d'une branche à une autre.

Restaurer un fichier depuis un commit spécifique avec un dialogue interactif

```
> git restore -s ID_du_commit --source nom_du_fichier
```

Contexte : Restaure un fichier depuis un commit spécifique en utilisant une interface interactive.

Afficher les différences dans un commit :

```
> git show ID_du_commit
```

Contexte : Affiche les différences introduites par un commit spécifique, y compris les fichiers modifiés et les lignes changées.

Ignorer les espaces lors de la fusion (merge)

```
> git merge --ignore-space-change nom_de_la_branche
```

Contexte : Fusionne deux branches en ignorant les changements d'espacement. Utile pour éviter les conflits mineurs dus à des différences d'espacement.

Générer un rapport de bogue

```
> git-bugreport
```

Collecte les informations permettant à l'utilisateur de déposer un rapport de bogue

Voir les révisions et l'auteur des changements

```
> git-blame
```

Montrer la révision et l'auteur qui ont modifié en dernier chaque ligne d'un fichier

Git rebase Vs Git reset Vs git revert

Cas d'utilisation

Git rebase

- Pour fusionner des branches de manière plus propre. Si vous avez fusionné deux branches et que vous souhaitez nettoyer l'historique, vous pouvez utiliser git rebase pour déplacer les commits de la branche fusionnée vers la branche principale.
- Pour corriger les conflits de fusion. Si vous rencontrez des conflits de fusion, vous pouvez utiliser git rebase pour résoudre les conflits un par un.
- Pour mettre à jour votre branche avec les modifications d'une autre branche. Si vous souhaitez mettre à jour votre branche avec les modifications d'une autre branche, vous pouvez utiliser git rebase pour fusionner les modifications de la branche cible dans votre branche actuelle.

Git reset

- Pour annuler des commits. Si vous souhaitez annuler un commit, vous pouvez utiliser git reset pour supprimer le commit de votre historique.
- Pour revenir à un état antérieur de votre projet. Si vous souhaitez revenir à un état antérieur de votre projet, vous pouvez utiliser git reset pour supprimer tous les commits après un certain point.

Git revert

- Pour annuler les modifications apportées par un commit. Si vous souhaitez annuler les modifications apportées par un commit, vous pouvez utiliser git revert pour créer un nouveau commit qui inverse les modifications du commit original

Qu'est-ce que GitHub ?

GitHub est une plateforme de développement logiciel en ligne qui permet aux développeurs de collaborer sur des projets. Elle offre une variété d'outils et de fonctionnalités qui facilitent le partage, la révision et la publication du code.

Fonctionnalités principales de GitHub

- **Contrôle de version** : GitHub utilise Git, un système de contrôle de version distribué. Git permet aux développeurs de suivre les modifications apportées au code au fil du temps.
- **Collaboration** : GitHub permet aux développeurs de travailler ensemble sur des projets en ligne. Les développeurs peuvent créer des branches, fusionner des modifications et discuter des changements.
- **Publication** : GitHub permet aux développeurs de publier leur code en ligne. Les développeurs peuvent créer des dépôts publics ou privés pour leur code.

Pull requests

Un pull request est une demande de fusion d'une branche dans une autre branche. Les pull requests sont utilisées pour collaborer sur des projets en ligne.

Comment créer un pull request

Pour créer un pull request, vous devez avoir accès aux deux branches impliquées dans la fusion. Vous pouvez créer une pull request en suivant ces étapes :

1. Branchez la branche que vous souhaitez fusionner dans la branche de destination.
2. Effectuez les modifications que vous souhaitez fusionner.
3. Committez les modifications.
4. Envoyez les modifications à GitHub.

Comment approuver un pull request

Pour approuver un pull request, vous devez avoir accès à la branche de destination. Vous pouvez approuver un pull request en suivant ces étapes :

1. Accédez à la page de la pull request.
2. Cliquez sur le bouton "Approuver".
3. Ajoutez un commentaire si vous le souhaitez.

Comment rejeter un pull request

Pour rejeter un pull request, vous devez avoir accès à la branche de destination. Vous pouvez rejeter un pull request en suivant ces étapes :

1. Accédez à la page de la pull request.
2. Cliquez sur le bouton "Rejeter".
3. Ajoutez un commentaire si vous le souhaitez.
- 4.

Forks

Un fork est une copie d'un dépôt GitHub. Les forks sont utilisés pour créer des versions dérivées d'un projet.

Comment créer un fork

Pour créer un fork, vous devez avoir un compte GitHub. Vous pouvez créer un fork en suivant ces étapes :

1. Accédez au dépôt que vous souhaitez forker.
2. Cliquez sur le bouton "Fork".
3. Entrez un nom pour votre fork.

Comment travailler sur un fork

Une fois que vous avez créé un fork, vous pouvez travailler dessus comme sur n'importe quel autre dépôt GitHub. Vous pouvez créer des branches, fusionner des modifications et publier votre code.

Comment fusionner un fork dans un dépôt parent

Pour fusionner un fork dans un dépôt parent, vous devez avoir accès au dépôt parent. Vous pouvez fusionner un fork en suivant ces étapes :

1. Branchez le fork dans le dépôt parent.
2. Effectuez les modifications que vous souhaitez fusionner.
3. Committez les modifications.
4. Envoyez les modifications à GitHub.

Les issues

Les issues sont une fonctionnalité de GitHub qui permet de suivre les problèmes et les demandes de fonctionnalités. Elles sont utilisées pour communiquer les problèmes aux développeurs et pour suivre leur progression.

Comment créer une issue

Pour créer une issue, vous devez avoir accès au dépôt GitHub dans lequel vous souhaitez créer l'issue. Vous pouvez créer une issue en suivant ces étapes:

1. Accédez au dépôt GitHub dans lequel vous souhaitez créer l'issue.
2. Cliquez sur le bouton "New issue".
3. Entrez un titre et une description pour l'issue.
4. Ajoutez des étiquettes si vous le souhaitez.
5. Cliquez sur le bouton "Create issue".

Comment suivre une issue

Une fois qu'une issue a été créée, vous pouvez la suivre en cliquant sur son lien dans le dépôt GitHub. Vous pouvez également ajouter des commentaires à l'issue pour discuter du problème ou de la demande de fonctionnalité.

Comment résoudre une issue

Si vous êtes le développeur responsable de résoudre une issue, vous pouvez le faire en suivant ces étapes :

1. Assignez-vous l'issue.
2. Modifiez le statut de l'issue en "En cours".
3. Effectuez les modifications nécessaires pour résoudre le problème.
4. Committez les modifications.
5. Envoyez les modifications à GitHub.

Une fois que vous avez résolu le problème, vous pouvez changer le statut de l'issue en "Résolu".

Types d'issues

GitHub propose différents types d'issues, notamment :

- Bug : Un bug est un problème qui empêche le code de fonctionner correctement.
- Feature request : Une demande de fonctionnalité est une demande d'ajout d'une nouvelle fonctionnalité au code.
- Documentation : Une issue de documentation est une demande de mise à jour ou de création de documentation.
- Testing : Une issue de test est une demande de test d'une nouvelle fonctionnalité ou d'une correction de bug.
- Other : Un autre type d'issue est utilisé pour suivre des problèmes ou des demandes qui ne correspondent pas aux autres types.

Autres fonctionnalités de GitHub

En plus des fonctionnalités principales et des pull requests, GitHub offre une variété d'autres fonctionnalités, notamment :

- Wikis : Les wikis sont utilisés pour créer des documentations et des guides.
- Discussions : Les discussions sont utilisées pour discuter du code et des projets.
- Marketplace : Le marketplace propose une variété d'outils et de services pour GitHub.